previous next contents top

A network file system over HTTP: remote access and modification of files and *files*

Abstract

The goal of the present HTTPFS project is to enable access to remote files, directories, and other containers through an HTTP pipe. HTTPFS system permits retrieval, *creation* and *modification* of these resources as if they were regular files and directories on a local filesystem. The remote host can be *any* UNIX or Win9x/WinNT box that is capable of running a Perl CGI script and accessible either directly or via a web proxy or a gateway. HTTPFS runs entirely in user space.

The current implementation fully supports reading as well as creating, writing, appending, and truncating of files on a remote HTTP host. HTTPFS provides an isolation level for concurrent file access stronger than the one mandated by POSIX file system semantics, closer to that of AFS. Both an API with familiar open(), read(), write(), close(), etc. calls, and an interactive interface, via the popular Midnight Commander file browser, are provided.

This present document combines a paper and a Freenix Track talk presented at a *1999 USENIX Annual Technical Conference*, June 6-11, 1999; Monterey, CA, USA.

Overview

Unlike NFS and AFS, HTTP is supported on nearly all platforms, from IBM mainframes to PalmPilots and cellular phones, with a widely deployed infrastructure of proxies, gateways, and caches. It is also regularly routed through firewalls. Using standard HTTP GET, PUT, HEAD and DELETE request methods, a rudimentary network file system can be created that runs cross-platform (e.g., Linux, Solaris, HP-UX, and Windows NT) on a variety of off-the-shelf HTTP servers: Apache, Netscape, and IIS. The HTTPFS can be used either programmatically or via an interactive interface.

HTTPFS is a user-level file system, implemented by a C++ class library on a client site, and a Perl CGI script on a remote site. The C++ framework of VNode, VNode_list, HTTPTransaction, MIMEDiscreteEntity etc. classes may be employed directly. Alternatively, HTTPFS functionality can be extended to arbitrary applications by linking with a library that transparently replaces standard file system calls (e.g., `open()`, `stat()`, and `close()`). This operation does not patch the kernel or system libraries, nor does it require system administrator privileges. The interposed functions invoke the default implementations, unless a file with an "`http://`" prefix is accessed. The HTTPFS client framework will handle the latter case. This permits URLs being used whenever a regular file name is expected, as an argument to `open()`, `fopen()`, `fstream()`, or a command-line parameter to file utilities. No source code needs to be modified, or even recompiled.

An important feature of HTTPFS is that it can provide a file-centric view of remote resources and containers that are not necessarily files or directories on a remote computer. Anything which an HTTPFS server can apply GET, PUT, DELETE methods to, and has timestamps and size attributes, may be accessed and manipulated as if it were a file. With HTTPFS, an off-the-shelf application may `open()`, `read()`, `write()` a "file" that may in reality be a database table, an element in an XML document, a property in the registry, an ARP cache entry, or the input or output of a process.

Borrowing from database terminology, HTTPFS provides an isolation level of "Repeatable Read" for concurrent file transactions. Once a process opens a file, it will not see changes to the file made by other concurrently running processes. This isolation is different from standard POSIX semantics, which provides for a "Dirty Read" isolation -- updates made to the file by other processes are visible before the file is closed. The difference in semantics is important, but only when a file is being concurrently read and modified. As was mentioned above, HTTPFS may permit a file-type access to a table of a relational database. In this particular case, the "Repeatable Read" isolation level is appropriate as it is the default for an ANSI-compliant database.

Usage Examples

In a separate document

Hypertext Transfer Protocol

HTTP is an application-level protocol for distributed, collaborative, hypermedia information systems

[HTTP]. It is a request/response protocol, where the client submits a request to the server, the server processes the request, and sends a response to the client. HTTP is open-ended, in that it allows new request/ response pairs to be defined. The message format is similar to that used by Multipurpose Internet Mail Extensions (MIME).

An HTTP transaction is in some sense a remote procedure call. An HTTP message specifies both an operation and the data on which to invoke the operation. The protocol provides facilities for exchanging data (arguments and results), and meta-data. The latter specialize a request and a response, carry authentication information and credentials, or annotate the content. Most HTTP transactions are synchronous, although HTTP/1.1 provides for asynchronous and batch modes. Furthermore, HTTP allows intermediaries (caches, proxies) to be inserted into the response-reply chain.

An HTTP request includes the name of the operation to apply and the name of the resource. Additional parameters if needed are communicated via request headers, or a request body. The request body may be an arbitrary stream of bytes. The HTTP/1.1 standard defines methods GET, HEAD, POST, PUT, DELETE, OPTIONS, and TRACE, which can be further extended by a particular server.

- The GET method retrieves the requested data along with some meta-information about the data.The data is denoted by a URI (universal resource identifier). The GET method can be conditional; if the resource has not been modified since the specified date, no data is returned. This form is useful when a cached copy of the resource exists.
- The HEAD method works similarly to the GET method, except that the server returns only the meta-data describing the properties of a resource.
- The PUT method stores the supplied data in the specified URI. Once PUT, the data will be available via a later GET.

HTTPFS maps these methods to the corresponding file access operations, while fully preserving the methods' semantics defined in the HTTP/1.1 document.

Of particular interest is the extensibility of the HTTP protocol. A client can submit arbitrary headers, which are available to the corresponding web server. The server may send arbitrary meta-data as response headers as well. In addition, a client and a server may exchange meta-information via "`name=value`" attribute pairs of the standard `Content-Type:` header.

Implementation of HTTPFS: Client

HTTPFS client is implemented by a C++ framework. It carries out HTTP transactions with a server and maintains a local cache of fetched files and directory listings. A file being opened for reading or modification is first fetched from a server in a GET transaction. However, if the file is already in cache, a conditional GET request is issued to verify that the cached copy is up-to-date, and reload it if not. When a file is being opened for writing, an additional `Pragma:` header is included in the GET request to inform the server of the open mode: `O_RDWR`, `O_WRONLY`, `O_CREAT`, `O_EXCL`, `O_TRUNC` or `O_APPEND`. The server may then create, truncate, or lock the resource. A response from the server is translated into the result of the `open()` call. Reads and writes to the opened file are then directed to the local copy. On `close()`, if the local copy has been modified, it is written back using PUT.

Status inquiries, e.g., `stat()`, `lstat()`, `readlink()`, etc., are implemented by submitting a HEAD request. A `Pragma:` request header tells the server which particular status information about the resource is requested.

Scanning of a directory -- `opendir()`, `readdir()`, `closedir()` -- is similar to accessing a file: a GET request is issued for a directory URI, and the resulting directory listing is locally cached.

Appendix A gives a detailed mapping between the file system API and HTTP requests and responses.

Implementation of HTTPFS: Server

A MCHFS server is one particular HTTPFS server. It is a Perl CGI script which executes HTTPFS requests and provides access to resources and containers. In the case of MCHFS, the resources and containers happen to be regular files and directories of a computer that runs this CGI script. The script thus lists directories on its own server, sends files, and accepts new content for old or newly created files.

According to a tradition, an HTTP server operates in a `chroot`ed environment. For example, when asked to retrieve a resource `http://hostname/README.html`, the server sends a file located at `$DOCUMENT_ROOT/README.html` (if exists), where `$DOCUMENT_ROOT` is something like `/opt/apache/htdocs`. MCHFS honors this convention:

```
open("http://hostname/cgi-bin/admin/
      MCHFS-server.pl/README.html", O_RDWR);
```

will let you access the same `$DOCUMENT_ROOT/README.html` file. Still MCHFS offers to escape the `chroot`ed confines and access files anywhere in its file system. This can be accomplished by using a distinguished path component `DeepestRoot`, which refers to the root of the server's file system. For example:

```
open("http://hostname/cgi-bin/admin/
       MCHFS-server.pl/DeepestRoot/etc/passwd", O_RDONLY);
open("http://hostname/cgi-bin/admin/MCHFS-server.pl/DeepestRoot/
       WinNT/Profiles/Administrator/NTusers.dat", O_RDONLY);
```

This is discussed further in the section on security considerations, below.

MCHFS allows any web browser to view directory listings and files. A directory request is returned as plain text, in a format similar to a '`ls -l`' listing. Because MCHFS understands regular GET requests, you can use a web browser to verify that MCHFS is installed and functioning properly. Any other user agent -- Wget or the plain telnet -- may be employed as well.

Transparent replacement of system calls

An application accesses the file system API either using low-level open/read/write/close calls, or via abstract file system interfaces (e.g., standard I/O, stream I/O, or ports). The latter are implemented, under the covers, through the open/read/write/close. Once these low-level functions are impersonated (and extended to handle `http://` "file names"), HTTPFS becomes available to any application without modifying the application's source code.

One does not need to patch the kernel or system libraries to intercept the POSIX filesystem API calls. One can do it safely, and without system administrator privileges by linking the application with

replacement versions of these low-level API functions. The recipe for doing so is as follows:

- compile a stub function with the name of the replaced routine;
- partially and statically link the stub with default implementations of the functions being intercepted;
- link the result with an application, the HTTPFS client library, and necessary standard libraries; the link mode may be either static, dynamic, or mixed.

Source code for an application is not required, only its object (compiled) form; the application need not be aware that it is using HTTPFS.

A web page [Intercept] explains this technique in detail, and discusses another use of this interception approach: implementing processes-as-files.

Pushing the envelope and security holes

The MCHFS script obviously opens up the file system of a host computer to the entire world. Furthermore, if a particular HTTPFS server chooses to interpret GET/PUT requests as output/input from an application (`sh` in particular), the whole system becomes exposed. Clearly this may not be desirable. Therefore, one may want to restrict access to MCHFS to trusted hosts or users. These authentication/ authorization policies are the responsibility of a web server's administrator; MCHFS need not be aware of them.

In addition, the MCHFS server may implement its own resource restriction policies. For example, it can refuse PUT requests, which effectively makes exported file systems read-only. MCHFS could permit modification or listing of only certain files, or disallow use of `DeepestRoot` and "`..`" in file paths, thus confining users to a limited part of the file system tree.

Related work

HTTPFS is similar to FTPFS, a virtual file system used by Midnight Commander, Emacs and KDE to access remote FTP sites. There is also a similarity to NFS. There are, however, a number of differences:

- HTTPFS operates through TCP channels using HTTP, a simple stateless reliable protocol. It is less resource-hungry than FTP.
- HTTPFS can talk to *any* host that runs an HTTP server and capable of executing a Perl CGI script. The client runs not only on Linux (which provides such niceties as `userfs`), but on Solaris and HP-UX as well.
- HTTPFS works transparently through firewalls, HTTP proxies and web caches.
- HTTPFS also stands to benefit from various caching, load-balancing and replication facilities that web gateways offer.
- HTTPFS can rely on authentication mechanisms already built into web servers (e.g., Secure HTTP), in addition to its own access control.
- HTTPFS can serve "files" and list "directories" that are created on the fly. In particular, HTTPFS permits browsing of a remote *database* as if it were a local filesystem.
- Whenever a remote file or directory get accessed or modified, HTTPFS can synchronously fire up triggers and run hooks. This is very difficult to accomplish with FTP.

See [Metcast] for a description of another data-distribution service that builds upon HTTP riches. Design of a Linux-specific HTTP-based filesystem, in the context of WebDAV, userfs and perlfs, is discussed in [HL].

Availability and installation

The MCFS/HTTPFS adapter distribution is freely available from a HTTPFS web page
http://pobox.com/~oleg/ftp/HTTP-VFS.html
The distribution archive contains the complete and self-contained source code for the server and the adapter, and an INSTALL document. A manifest file tells what all the other files are for.

I have personally run the MCHFS on HP-UX and SunSparc/Solaris with Netscape and Apache HTTP servers, and on Windows NT running IIS. The HTTPFS client -- the MC/HTTPFS adapter in particular -- ran on Sun/Solaris, HP-UX, and Linux platforms. The adapter successfully communicated with a Midnight Commander on a Linux host (MC version 4.1.36, as found in S.u.S.E. Linux distribution, versions 5 and 6).

I have not yet implemented the `unlink()`, `rename()`, `mkdir()`, and `chmod()` file system calls. I should also look into persistent HTTP connections and an option of transmitting only selected pieces of a requested file, which HTTP 1.1 allows (and encourages).

Summary: the OS *is* the browser

This article presents a poor-man's network file system, which is simple, very portable, and requires the least privileges to set up and run. HTTPFS offers a glimpse of one of *Plan9*'s jewels -- a uniform file-centric naming of disparate resources -- but without Plan9. This file system showcases HTTP, which is capable of far more than merely carrying web pages. HTTP can aspire to be the kingpin protocol that glues computing, storage, etc. resources together to form a distributed system -- the role *9P* plays in [Plan9].

The design of HTTPFS suggests that, contrary to a cliche, it is the OS that is the browser. While Active Desktop lets you view local files and directories as if they were web pages, HTTPFS allows access to remote web pages and other resources as if they were local files. HTTPFS has all the attributes of an OS component: it implements (a broad subset of) the filesystem API; it maintains "vnodes" and "buffer caches"; it interacts with a persistent store and offers a uniform file-centric view of various remote resources. On the other hand, HTTPFS provides a superset of remote access services every Web browser has to implement on its own. The HTTPFS and other local and network filesystems manage storage and distribution of content, while an HTML formatter along with `xv`, `ghostscript` and similar applications provide interpretation and rendering of particular kinds of data. Thus as far as the OS is concerned, viewing a web page is to be thought similar to displaying an image file off an NFS-mounted disk, and searching the Web is no different than running `glimpse` on a local filesystem.

References

[HTTP] *HTTP Version 1.1*, R. Fielding, J. Gettys, J. Mogul, H. Frystyk Nielsen, and T. Berners-Lee, January 1997. RFC-2068

[Intercept] *Patch-free User-level Link-time intercepting of system calls and interposing on library functions*, Oleg Kiselyov <http://pobox.com/~oleg/ftp/syscall-interpose.html>

[MC] *The Midnight Commander* <http://www.gnome.org/mc/>

[Metcast] *Pushing Weather Products via an HTTP pipe. Introduction to Metcast*, Oleg Kiselyov <http://zowie.metnet.navy.mil/~spawar/JMV-TNG/>

[HL] *An HTTP filesystem for Linux?* <http://rufus.w3.org/linux/httpfs/>

[Plan9] *Plan 9 from Bell Labs*, Rob Pike, Dave Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Trickey, Phil Winterbottom <http://plan9.bell-labs.com/plan9/doc/9.html>

Acknowledgement

Comments, suggestions, and shepherding by Chris Small are greatly appreciated.

Last updated June 25, 1999

This site's top page is **http://pobox.com/~oleg/ftp/**

ITjungle
Series-Adapt-Thrive

Going crazy?

Newsletters Subscriptions Advertise About Us Contact Search Home

Midrange Programmer
How-To Advice & Free Code

OS/400 Edition
Volume 1, Number 10 -- May 23, 2002

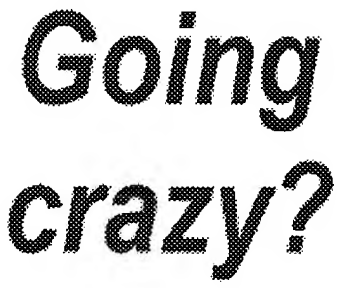
JavaScript and the SHELL Command

by Shannon O'Donnell

If you're an RPG programmer and you're just now getting into writing HTML, chances are you've discovered a lack of functionality that you used to have in RPG--specifically, the ability to execute commands or call other programs from HTML. Because HTML runs in a browser, and because of the inherent, built-in security of browsers, it's difficult to call other programs from within HTML code. But sometimes you need to. What's the solution? Java Script and SHELL!

The SHELL Command

Visual Basic and C++ programmers have known about the SHELL command for years. SHELL is a Microsoft Win32 API that allows Windows programmers to execute commands from a Shell or command line. The SHELL API, like all Win32 APIs, is not easy to use--at least not for non-Windows programmers. However, you don't have to understand all of its syntax and features in order to use it. You just have to know how to use it.



In this article, I'll show you how to use the SHELL API from within HTML to perform several common functions. Since you want to know more about how to actually use it from your own HTML applications, and probably couldn't care less about the syntax and background of the SHELL command, I'm going to skip the history lesson. If you want to learn more about the syntax of the SHELL command, take a look at the Microsoft's Web site. Just search on *SHELLEXECUTE*, and you'll find pages and pages about the SHELL API, its syntax, and how to use it.

BACK ISSUES

SHELL is an ActiveX API. That is, SHELL has certain properties and methods that are exposed, or are public, which you can access from your own applications. SHELL also follows the Component Object Model, or COM, method of access, which means, among other things, that it has certain standard behaviors. You can take advantage of these standard behaviors in your own HTML applications. To do so, you'll need to write a little bit of JavaScript.

JavaScript and SHELL

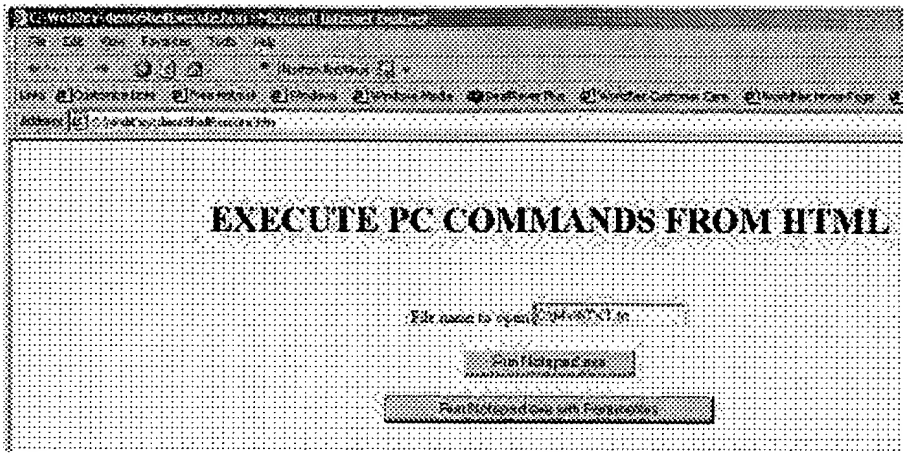
If you've read some of the other JavaScript articles in *Midrange Programmer*, you're probably pretty proficient at using JavaScript by now. If you're not, you may want to go back and read those articles again, just to get familiar with it. For this article, we're going to write some JavaScript in order to demonstrate how to use the SHELL ActiveX object to execute commands on your PC.

To start, open up Microsoft Notepad.exe and create a new HTML document.

```
<HTML>
<HEAD>

</HEAD>
<BODY>
<FORM name="Form1">
<CENTER>
<BR><BR>
<H1>Execute PC Commands From HTML </H1>
<BR><BR>
<File Name to Open:> <Input type="text"
   name="filename"/>
<BR><BR>
<Input type="Button" name="Button1"
   value="Run Notepad.exe" />
<BR><BR>
<Input type="Button" name="Button2"
   value="Run Notepad.exe with Parameters" />
</CENTER>
</BODY>
</FORM>
</HTML>
```

Save this document with a name like *DemoSHELLAPI.htm* and then open it in a browser. It should look something like the one shown here:



At this point, your HTML code doesn't do anything, because you haven't added any code behind it. Let's do that now.

Adding the Power

To use the SHELL API, you'll need to write a JavaScript function and define the SHELL API parameters within it. And you'll need to execute the function for when the user clicks a button.

Modify the DemoSHELLAPI.htm file to add the following code between the opening and closing HEAD tags:

```
<SCRIPT type="text/javascript" LANGUAGE="JavaScript">
  function executeCommands(inputparms)
   {
  // Instantiate the Shell object and invoke
   its execute method.

    var oShell = new ActiveXObject("Shell.Application");

    var commandtoRun = "C:\\Winnt\\Notepad.exe";
    if (inputparms != "")
     {
      var commandParms = document.Form1.filename.value;
     }

 // Invoke the execute method.
     oShell.ShellExecute(commandtoRun, commandParms,
          "", "open", "1");
  }
</SCRIPT>

Modify your buttons, in the BODY of the text to add the
   "onClick" event to each.

<input type="Button" name="Button1"
   value="Run Notepad.exe" onClick="executeCommands()" />

<input type="Button" name="Button2"
   value="Run Notepad.exe with Parameters"
      onClick="executeCommands(' + hasPARMS + ')" />
```

What That Does

Here's what this JavaScript does.

First, it accepts a parameter from whatever HTML element calls it. In this case, the JavaScript is executed when the user clicks one of the buttons on the HTML form.

Next, the JavaScript will instantiate the ActiveX object *Shell.application*. This is the SHELL API. The instantiated object is then assigned to the variable named *oShell*. Incidentally, you can instantiate most ActiveX objects from within HTML in this manner.

Next, we'll set the value *commandtoRun* to the value of the command we want to execute, in this case *Notepad.exe*. Notice that there are two backslashes (\\) between each path parameter. This is required because, if you only used a single backslash in the path, JavaScript would interpret that single script as an operator. By making the command you wish to execute a variable, you can easily modify your HTML code so that you execute whatever command the user chooses from your HTML form. This will give you a lot of flexibility in your applications.

Next, if the user clicked the button to run NotePad.exe and to pass it a parameter of the name of a file to open, then the variable *inputparms* will contain the value *hasParms*, which is defined in the "onClick" event of the second button on this Form. If *inputparms* is not blank, the script will retrieve the value the user entered in the input field on the form and set the variable *commandParms* equal to that value.

Finally, we'll execute the command to run, Notepad.exe, by calling the ShellExecute method of the SHELL API. We'll pass the ShellExecute method several parameters, each individually enclosed within double quotation marks ("). The first parameter is the command to be executed, in this case Notepad.exe. The second parameters are the input parameters, if any, to the command. The other parameters are not important to this discussion, but if you're interested in using them, be sure to check out the SHELL documentation on Microsoft's Web site.

Save this file and reopen it in your browser and try it out.

Powerful Functions

Using ActiveX objects from within HTML gives you the power and flexibility you need to code professional and useful applications. Explore the SHELL API and some of the other Win32 APIs and see how you can expand on them to add even more power to your applications.

Development

In an effort to help the most patients, doctors are always looking for the quickest and easiest way to get things done. At Iredell Memorial Hospital in Statesville, NC, the green screen application that accessed patient records and information was just not good enough. Scott Philemon was charged with the task of bringing this valuable information to a browser-based system and he found that AVR was the best way for him to do that. He has created a simple point-and-click system that is intuitive rather than the cumbersome green-screen system with which most doctors were uncomfortable. The most important feature of this new application is that doctors will be able to connect, via a modem, to this application and not have to make the trip into the hospital to check on the status of patient blood tests and other medical procedures.

"I was given this project and told to just find the best way to get it done. I heard java, java, java all the time so I took a couple of introductory classes and realized that I would spend a year just learning it before I would even be able to start on this project. With AVR I can really use my RPG skills to create browser-based applications. The transition was so easy and I'm so pleased with the results that I'm recommending that we do ONLY Web applications in the future."

—Scott Philemon, Iredell Memorial Hospital

ASNA Visual RPG (AVR) for Web, Windows and .NET Development

ASNA Visual RPG (AVR) is an integrated development environment for creating enterprise Web, Windows and .NET applications. Transparent database access; an integrated editor, compiler and debugger; support for emerging standards such as XML and SOAP; and equally powerful Web or Windows deployment possibilities make ASNA Visual RPG the one application development environment you can't afford to ignore! Use your RPG skills to develop Web, Windows and .NET applications today.

Download your FREE trial of AVR today!

http://www.asna.com/downloads.asp

Miscellaneous Quirks and Workarounds

Direct API Calls

To reiterate the problem of scripting the unscriptable: if there is no external interface designed for something, how do you get to it?

As a matter of fact, there are interfaces to almost everything on Windows: the APIs. Unfortunately, you can't call an API from script.

One workaround is to write your own API wrappers as ActiveX components - a speedy way of doing it if you have the requisite knowledge of the APIs, and have a VB or C compiler and can use them. I've done that with my XAPI and SOX DLLs.***[In transition, link removed... sorry folks!]***

Another way to do this -a more "generic' approach - is an ActiveX DLL that will act as a wrapper for any API. This is much more work, and has been implemented by Jim Warrington in his **WSHATO object**.

A third way, which is fraught with security perils, is to do API calls by proxy - something scripting is quite capable of doing. Thus we have:

API Calls From Script Via Microsoft Office

The following example is in the form of a WSF file - save it as, say, APICallViaXL.wsf and run it. Note that this may require temporarily lowering Microsoft Office's macro security settings. After you do it, this should also motivate you to push them right back up, since if you can poke around with a system API from WSH this way, so can any other script that runs on the system.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<package>
<job>
<object id="xl" progid="Excel.Application"/>
<script language="VBScript">
<![CDATA[
' Started Excel above
xl.Visible = True
' Add a new workbook
Set xlBk = xl.Workbooks.Add
' Add a module
Set xlMod = xlBk.VBProject.VBComponents.Add(1)
' Add a macro to the module...
strCode = Getresource("macro")
xlMod.CodeModule.AddFromString strCode
' Run the new macro!
iData = xl.Run("Uptime")
Set xlMod = Nothing
xlBk.Saved = True
xl.Quit
wscript.echo f
]]>
```

```
</script>
<resource id="macro">
<![CDATA[
Declare Function GetTickCount Lib "kernel32" () As Long
Function Uptime()
Uptime = GetTickcount
End Function
]]>
</resource>
</job>
</package>
```

The Oddities of ByVal and ByRef

This is not strictly a Rube Goldberg solution; I'm just documenting the quirks in how the ByVal and ByRef calls in VBScript work. Basically, they are an absolute mess in terms of order due to requirements for backwards compatibility. Eric Lippert of the Windows Scripting team covered this in excruciating detail in a post to a scripting newsgroup in 1999, and I am simply linking to **a local copy of his post**.

WSH
https